# Studienarbeit Embedded Linux


## Peter Hüwe

<ph@huewe.info>



version 0.1, Januar 2009, License: CC-by-sa


## Inhaltsverzeichnis

# 1. Zusammenfassung

Diese Studienarbeit zum Fach Embedded Linux, gehalten von Herrn Prof. Dr. Högl im Wintersemester 08/09, beschreibt eine kleine Auswahl meiner Tätigkeiten und Versuche die ich im Umfeld von Embedded Linux in diesem Semester unternommen habe — Aufgrund meines starken persönlichen Interesses an diesem Themengebiet sind im Laufe des Semesters einige Projekte bearbeitet worden, von denen ich hier leider nicht alle im vollen Detail behandeln kann - falls der Leser noch Fragen zu dem ein oder anderen Thema hat, so genügt eine kleine Email an mich (ph@huewe.info).

Dieses Dokument steht unter der Creative Commons Attribution Share-Alike 3.0 License und kann somit unter den Lizenzbedingungen frei verwendet werden (CC by-sa).

Insbesondere die Veröffentlichung durch Herrn Prof. Dr. Högl ist explizit erlaubt.

Außerdem würde ich mich über eine kurze Notiz bei Verwendung dieser Arbeit sehr freuen.

# 2. Taihu-Board

Das Taihu-Board ist ein von AMCC erstelltes Referenz-Design zum Evaluieren des 405EP PowerPC Prozessors.

Das Board verfügt unter anderem über folgende Ausstattung:

- 333 MHz AMCC 405EP processor
- 128 MByte SDRAM
- 32 MByte Flash
- 2 MByte boot Flash
- 2x 10/100 Ethernet ports
- 1 USB 1.1 port (device)
- JTAG connector
- Trace connector
- PCI host connector (33/66 MHz)
- Cardbus connector (33 MHz)
- LCD-Display 2x16 Zeichen, 80 Zeichen DDRAM

Weitere Informationen und Ressourcen zum Taihu-Board finden sich im Download Portal von AMCC.

Das Board wurde Herrn Högl kostenlos von AMCC zur Verfügung gestellt.

Dieses Board ist insbesondere interessant, da es bereits zu weiten Teilen von Linux unterstützt wird, mit Ausnahme des LCD-Displays — für dieses existierte bisher kein Linux-Treiber.

## 2.1. Aufbau und Entwicklungsumgebung

Das Board ist meinem Entwicklungsrechner über die serielle Schnittstelle und per Ethernet verbunden — es empfiehlt sich für die Entwicklung auf einem Board immer ein getrenntes Netz zu verwenden um das normale Netz nicht zu stören.

Ein Problem bei der Inbetriebnahme des Taihu-Boards war jedoch eine passende Spannungsquelle zu finden - mein normales Schalternetzteil liefert leider nur 300mA, während das Taihu-Board laut Handbuch mindestens 3A erwartet - diese Stromspitze wird beim Initialisieren des RAMs erreicht, liefert das Netzteil zuwenig Strom schlägt der built-in RAM-Test fehl und das Board startet neu.

Um das Board dennoch mit relativ sauberen 5V zu versorgen habe ich anschließend einen der Molex(tm)-Stecker vom Netzteil meines Desktop-Rechners herausgeführt und dort den Adapter meines Stecker-Netzteiles aufgesteckt, sodass im Inneren des Zentralsteckers 5V und außen das Massepotential anliegt.

Auch wenn diese Vorgehensweise funktioniert, wäre für den den Dauereinsatz ein passendes Netzteil dringend zu empfehlen.

Nachdem das Board entsprechend verkabelt war wurde der erste Versuch unternommen mit dem Board zu kommunizieren - welcher natürlich prompt fehl schlug, denn leider unterstützt das Board an seiner RS232 Schnittstelle weder Handshake noch Flusskontrolle - dies äußerte sich so, dass jeweils nach ein paar Zeichen das RS232-Terminal Programm (in meinem Fall `kermit`) einfror und keine weiteren Eingaben akzeptierte.

Erst durch das Setzen der beiden Optionen `set handshake none` und `set flow-control none` funktionierte die Kommunikation mit dem Board über die serielle Schnittstelle ohne weitere Probleme.

### Konfiguration von Kermit

Damit man nicht bei jedem Aufruf von kermit die Kommunikationsparameter jedesmal neu setzen muss legt man in seinem Homeverzeichnis eine Datei namens `.kermrc` an und hinterlegt in dieser die Parameter.

Für das Taihu-Board sieht meine Konfiguration folgendermaßen aus:

```
set line /dev/ttyUSB0
set speed 9600
set carrier-watch off
set handshake none
set flow-control none
robust
set file type bin
set file name lit
set rec pack 1000
set send pack 1000
set window 5
```

Anschließend wurden in U-Boot mittels `printenv` und `setenv` diverse Parameter ausgelesen und gesetzt und anschließend ein paar der eingebauten Test- und Demo-Programme ausprobiert.

Nachdem dies abgeschlossen war, wurde das Board durch einen kurzen Druck auf den Reset-Schalter neu gestartet und nach kurzer Wartezeit erschien auch schon der Linux Login-Prompt.

## 2.2. ELDK und erste Versuche mit der Toolchain

Auf der Ressource-CD des Boards befand sich neben den Datenblättern und diverser Software auch das sogenannte "Embedded Linux Development Kit" kurz `ELDK` von denx.de. Dieses beinhaltet eine komplette, vorkonfigurierte Toolchain die zur Entwicklung von Software für PowerPC Boards eingesetzt werden kann - nach kurzer Installation und der Anpassung der Shellvariablen war die Toolchain einsatzbereit und das erste Programm konnte entwickelt werden: - ein einfaches `Hello World` Programm demonstrierte, dass die Toolchain funktionierte.

### 2.2.1. I2C Tools

Um die Zusammenarbeit der Toolchain mit Makefiles zu erforschen und um die diversen Chips auf dem I2C-Bus auszulesen wurden nun die I2C Tools aus dem lm-sensors Projekt kompiliert und anschließend per ftp übertragen:

```
wget http://dl.lm-sensors.org/i2c-tools/releases/i2c-tools-3.0.2.tar.bz2
tar -xvjf i2c-tools-3.0.2.tar.bz2
cd i2c-tools-3.0.2
ARCH=ppc CROSS_COMPILE=ppc_4xx- make CC=ppc-linux-gcc prefix=/tmp/i2ctools
ARCH=ppc CROSS_COMPILE=ppc_4xx- make CC=ppc-linux-gcc prefix=/tmp/i2ctools install
ncftpput -R 192.168.1.62 / /tmp/i2ctools
```

Auf der Board-Seite musste noch ein passendes I2C-Device angelegt werden und die Tools mussten außerdem das Ausführrecht erhalten. Anschließend konnte ich mit `i2cdetect` den Bus scannen und mit `i2cdump` mir die Werte der gefundenen Chips anzeigen lassen - in Code sieht dies dann folgendermaßen aus:

```
/var/ftp/i2ctools/sbin # cd /var/ftp/i2ctools/
/var/ftp/i2ctools # cd /var/ftp/i2ctools/sbin/
/var/ftp/i2ctools/sbin # chmod a+x *
/var/ftp/i2ctools/sbin # mknod /dev/i2c-0 c 89 0                          (1)
/var/ftp/i2ctools/sbin # ./i2cdetect 0                                   (2)
WARNING! This program can confuse your I2C bus, cause data loss and worse!
I will probe file /dev/i2c-0.
I will probe address range 0x03-0x77.
Continue? [Y/n]
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- 49 -- -- -- -- -- --
50: 50 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- --

/var/ftp/i2ctools/sbin # ./i2cdump 0 0x49                                 (3)
No size specified (using byte-data access)
WARNING! This program can confuse your I2C bus, cause data loss and worse!
I will probe file /dev/i2c-0, address 0x49, mode byte
Continue? [Y/n] y
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f    0123456789abcdef
00: 27 00 4b 50 50 50 50 50 50 50 50 50 50 50 50 50    '.KPPPPPPPPPPPPP
10: XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX    XXXXXXXXXXXXXXXX
20: XX XX 4b XX XX XX XX XX XX XX XX XX XX XX XX XX    XXKXXXXXXXXXXXXX
30: XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX    XXXXXXXXXXXXXXXX
40: 27 00 4b 50 50 50 50 50 50 50 50 50 50 50 50 50    '.KPPPPPPPPPPPPP     (4)
50: XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX    XXXXXXXXXXXXXXXX
60: XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX    XXXXXXXXXXXXXXXX
70: XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX    XXXXXXXXXXXXXXXX
80: XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX    XXXXXXXXXXXXXXXX
90: XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX    XXXXXXXXXXXXXXXX
a0: XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX    XXXXXXXXXXXXXXXX
b0: XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX    XXXXXXXXXXXXXXXX
c0: XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX    XXXXXXXXXXXXXXXX
d0: XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX    XXXXXXXXXXXXXXXX
e0: XX XX XX XX XX XX XX XX XX XX XX XX XX XX 50 XX    XXXXXXXXXXXXXXPX
```

```
f0: XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX    XXXXXXXXXXXXXXXX


/var/ftp/i2ctools/sbin # ./i2cdump 0 0x49
No size specified (using byte-data access)
WARNING! This program can confuse your I2C bus, cause data loss and worse!
I will probe file /dev/i2c-0, address 0x49, mode byte
Continue? [Y/n] y
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f    0123456789abcdef
00: 28 00 4b 50 50 50 50 50 50 50 50 50 50 50 50 50    '.KPPPPPPPPPPPPP     (5)
10: XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX    XXXXXXXXXXXXXXXX
20: XX XX 4b XX XX XX XX XX XX XX XX XX XX XX XX XX    XXKXXXXXXXXXXXXX
30: XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX    XXXXXXXXXXXXXXXX
40: 27 00 4b 50 50 50 50 50 50 50 50 50 50 50 50 50    '.KPPPPPPPPPPPPP
50: XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX    XXXXXXXXXXXXXXXX
60: XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX    XXXXXXXXXXXXXXXX
70: XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX    XXXXXXXXXXXXXXXX
80: XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX    XXXXXXXXXXXXXXXX
90: XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX    XXXXXXXXXXXXXXXX
a0: XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX    XXXXXXXXXXXXXXXX
b0: XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX    XXXXXXXXXXXXXXXX
c0: XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX    XXXXXXXXXXXXXXXX
d0: XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX    XXXXXXXXXXXXXXXX
e0: XX XX XX XX XX XX XX XX XX XX XX XX XX 50 XX XX    XXXXXXXXXXXXXPXX
f0: XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX    XXXXXXXXXXXXXXXX

/var/ftp/i2ctools/sbin # ./i2cdump 0 0x50
No size specified (using byte-data access)
WARNING! This program can confuse your I2C bus, cause data loss and worse!
I will probe file /dev/i2c-0, address 0x50, mode byte
Continue? [Y/n] y
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f    0123456789abcdef
00: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff    ................     (6)
10: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff    ................
20: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff    ................
30: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff    ................
40: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff    ................
50: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff    ................
60: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff    ................
70: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff    ................
80: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff    ................
90: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff    ................
a0: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff    ................
b0: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff    ................
c0: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff    ................
d0: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff    ................
e0: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff    ................
f0: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff    ................
```

1. Das I2C-Dev Interface hat die Major-Nummer 89 und ist ein char-Device.
2. Scanne den I2C-Bus Nummer 0
3. Bei Adresse 0x49 wurde etwas gefunden - dies zeigen wir nun mit i2cdump an

4. Binärdaten - hier hilft nur das Datenblatt weiter
5. Werte die sich ständig verändern weisen auf (Temperatur-)Sensoren hin.
6. Lauter gleichartige Einträge weisen auf ein EEPROM hin.

### User's Manual und Datasheets

Da man nun bei den Rohdaten angekommen ist, ist ein Blick in das User's Manual unvermeidlich - dieses findet sich auf der Ressource CD unter `/docs/eval_board/Taihu405EP_um_1.01.pdf`

Dort wird unter Punkt 1.12 und unter Punkt 1.13 beschrieben, welche Geräte am I2C-Bus angeschlossen sind und unter welchen Adressen diese ansprechbar sind:

| Adresse | Bauteil |
|---|---|
| 0x49 | Temperatur Sensor DS1755R |
| 0x50 | Serielles EEPROM - 2048 Bit |
| 0xXX | Weitere I2C Geräte, die von außen an den Bus angeschlossen werden können. |

Das Datenblatt des Temperatur Sensors erklärt relativ genau, welche Werte welche Bedeutung haben - um die Temperatur auszulesen, ist lediglich das erste Hex-Bytes von Bedeutung - dieses müssen wir nun noch umrechnen und interpretieren: `27 = +0010 0111` = $2^5 + 2^2 + 2^1 + 2^0$ (Laut Datenblatt) = 32 + 4 + 2 + 1 = 39° Celsius

Der Temperatursensor befindet sich auf der Unterseite des Boards, nahe der CPU - durch eine externe Wärmequelle in der Nähe konnte die Funktionalität getestet werden.

Der Einsatz von I2C-Tools birgt gewisse Risiken, insbesondere wenn man auf den Bus schreibt, sollte man sicher sein, dass man weiß was man tut - in der Regel ist auch das SPD des RAM per I2C erreichbar, ein falscher Wert dort und das Board bootet nicht mehr.


Die Unterstützung für I2C-Dev muss im Kernel aktiviert sein, damit die I2C-Tools funktionieren - die dazugehörige Konfigurationsoption heißt: `CONFIG_I2C_CHARDEV`

Die passenden Char-Devices sollten durch `udev` beim Laden

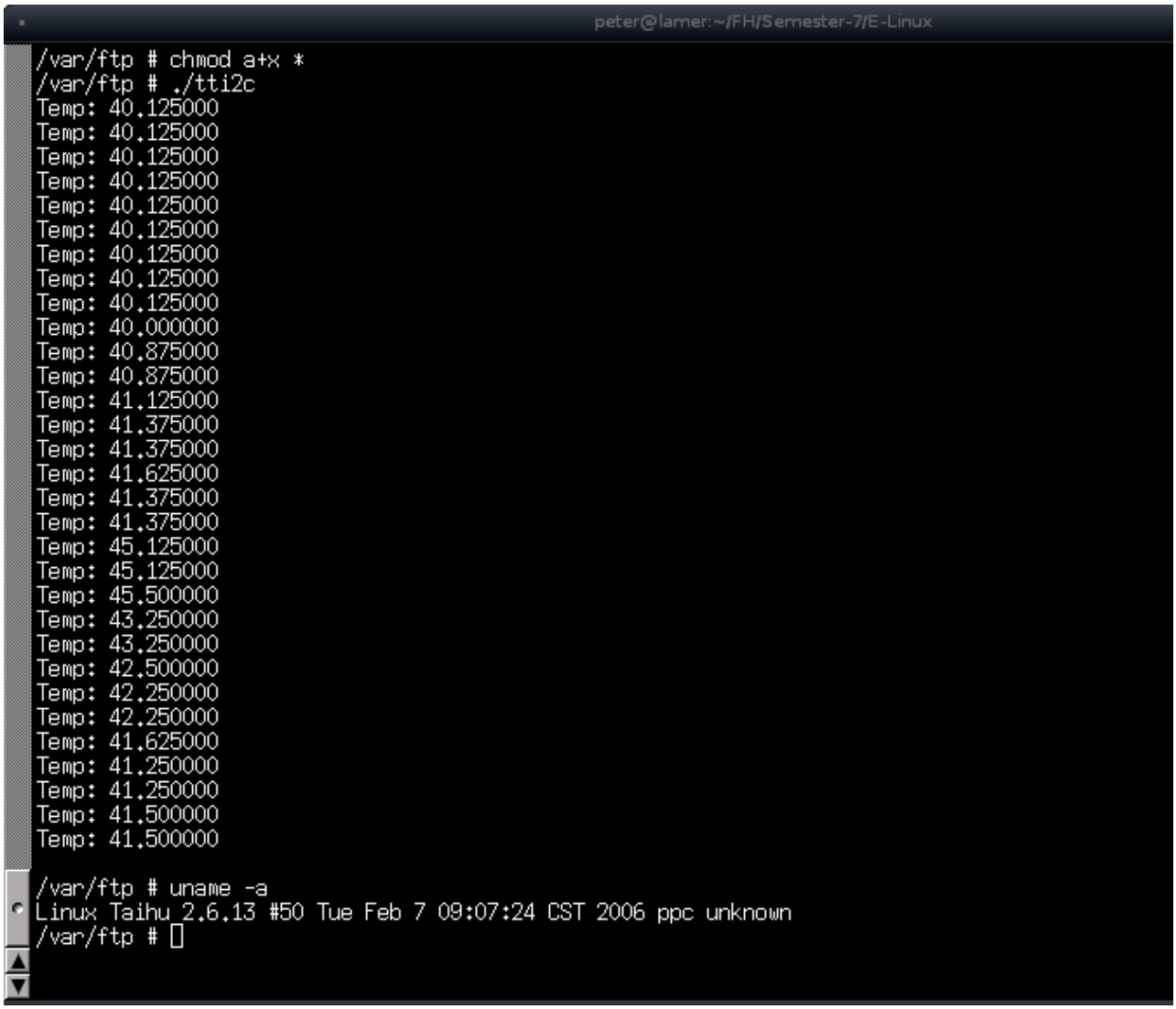
des Modules theoretisch automatisch erstellt werden - im obigen Szenario stand `udev` bzw. `mdev` noch nicht zur Verfügung.

### Weitere Details zu den I2C-Tools

Mithilfe der I2C-Tools lassen sich relativ einfach Applikationen schreiben, die direkt auf den I2C-Bus zugreifen - hierzu liefert die I2C-Tools Distribution ein passendes Header-File mit: `i2c-dev.h`.

Wer jedoch lieber mit `Python` statt mit C programmiert findet im Ordner `py-smbus` die passende Schnittstelle. Der Zugriff aus Python heraus ist sehr komfortabel und spart einem sehr viel Zeit.

Nachdem der erste Versuch die Temperatur auszulesen erfolgreich verlief, wurde nun noch eine kleine Applikation in C geschrieben, die — mithilfe der I2CTools — zuerst die Genauigkeit des Sensors auf 11-Bit setzt und anschließend jede Sekunde den Temperatursensor ausliest und das Ergebnis auf dem Bildschirm ausgibt.




Die Temperaturschwankungen wurden auch diesesmal durch eine externe Wärmequelle erzeugt.

### 2.2.2. Cardbus / PCMCIA

Das Taihu-Board verfügt neben vielen anderen Schnittstellen auch über ein Cardbus Interface für PCMCIA Karten.

Um zu testen in wie weit diese Schnittstelle bereits von Linux unterstützt wird, wurde der Slot kurzerhand mit einer Wlan Karte (Ralink Chip) bestückt - in der Ausgabe von `dmesg` zeigten sich bereits erste Anzeichen, dass die Karte erkannt wurde und lediglich noch die passenden Treiber für den Chip fehlten.

Diese wurden prompt von http://rt2x00.serialmonkey.com/ heruntergeladen und mit den üblichen Parametern kompiliert und auf das Board geladen.

Nachdem Treiber geladen war, meldete sich die Karte ordnungsgemäß - und lies sich mittels `iwconfig` konfigurieren, so dass nach wenigen Minuten

bereits eine Wlan Verbindung hergestellt war.


### 2.2.3. Kompilieren eines eigenen Kernels und verändern des Image

Nachdem die I2C-Tools erfolgreich getestet wurden, war es nun an der Zeit einen eigenen Kernel zu kompilieren und anzupassen, die I2C-Tools und die kleine Temperatursensor-Anwendung mit in das Root-Filesystem zu integrieren, sowie diverse weitere Modifikationen am Root-Filesystems durchzuführen.

Das Kompilieren des Kernels stellte sich als sehr einfach heraus, man musste lediglich darauf achten, dass man genauso wie bei den I2C-Tools die Umgebungsvariablen für das Cross-Compilieren richtig setzt und dass man einen gepatchten Kernel von denx.de verwendet — Normale Vanilla-Kernel von kernel.org sind nicht auf das Board angepasst und funktionieren deshalb nicht - auch dies wurde ausprobiert.

Nützliche Abweichungen von der `taihu_defconfig` sind unter anderem NFS-Support und I2C-Dev.

Das Bearbeiten des Root-Filesystem wird auf der ELDK-Seite How To Add Files sehr genau beschrieben.

Für mein Board habe ich folgende Änderungen vorgenommen:

- Einen Eintrag `/etc/fstab` um einen Eintrag für das sysfs erweitert: `none /sys sysfs defaults 0 1`
- Selbstkompilierte Busybox mit Unterstützung für mdev
- Hinzufügen selbst ersteller Programme

Durch sysfs und mdev, kann das Taihuboard anschließend dynamisch per Hotplug-Mechanismus die benötigten Device-Knoten erstellen - dies wird später bei der Beschreibung des LCD-Display Treibers noch eine Rolle spielen.

### 2.2.4. Flashen mit U-Boot

Das anschließende Flashen geht nach einer weiteren Anleitung (Linux in Flash) auch ohne Probleme - hierbei muss man natürlich aufpassen, dass man die richtigen Werte für sein Board verwendet und nicht ausversehen den Speicherbereich von U-Boot überschreibt.

## 2.3. Treiber für LCD-Display

Da bisher kein Linux-Treiber für das LCD-Display existierte, war es meine Aufgabe einen passenden Treiber dafür zu entwickeln - dabei sollte folgende Funktionalität abgebildet werden:

- Einfacher Schreibzugriff
- Schnittstelle für Rohdaten und -Kommandos
- Schnittstelle zum Ein- und Ausschalten der Hintergrundbeleuchtung

Der Treiber soll natürlich unter der GPL Lizenz veröffentlicht werden.

### 2.3.1. Recherche und Vorbereitung

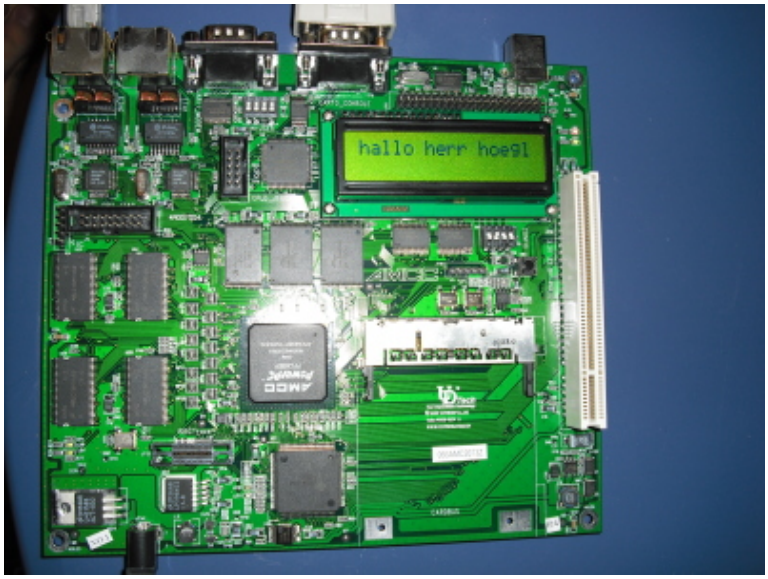
Bei der Suche nach Informationen über das LCD-Display, sind mir in U-Boot die Programme `lcd_cls`, `lcd_putc`, `lcd_cur`, und `lcd_puts` aufgefallen - mit diesen Programmen kann man das LCD-Display aus U-Boot heraus ansteuern. Da U-Boot auch als Open Source veröffentlicht und vertrieben wird, hatte ich somit schon relativ schnell einen groben Ansatzpunkt wie ich das LCD-Display ansteuern kann.

Außerdem fanden sich auf der Ressource CD noch 2 Datenblätter zum LCD-Display mit der Bezeichnung LCM1602 - jedoch beide auf Chinesisch! Jedoch wird in einer Fußnote, in Englisch, der dahinterliegende Controller S6A0069 von Samsung erwähnt. Für diesen Controller findet wiederum ein Datenblatt, welches die Ansteuerung relativ genau erklärt.

Dieser Controller besitzt die gleiche Funktionsweise wie der Sitronix ST7066U und der Hitachi HD44780.

### 2.3.2. Erster Erfolg

Durch meine bisherige Erfahrung beim Erstellen von Linux-Kerneltreibern dauerte es nicht sehr lange, bis der erste Text auf dem Display erschien - dabei handelte es sich um einen hardcodierten String, der Zeichen für Zeichen auf das Display geschrieben wurde - das Ergebnis kann man auf folgender Abbildung bewundern:




### 2.3.3. Schreiben und Anhängen durch Dateioperationen

Nachdem auch hier der erste Test geglückt war, wurde der bestehende Treiber um eine Character Device Schnittstelle erweitert - dadurch ist es möglich, dass jedes beliebige Programm einfach auf das LCD-Display schreiben kann, genauso wie in jede andere gewöhnliche Datei auch.

Durch das (fast) "Alles ist eine Datei" Konzept von Linux ist sogar einfaches Shell-Pipelining möglich.

Die wichtigsten Schritte hierbei waren das alloziieren der Major Nummer mittels `alloc_chrdev_region` und `MAJOR` sowie die Erweiterung der Dateioperationen, der sogenannten `fops`, um eine Open und eine Write Methode.

Insbesondere bei der Open-Methode bestand die Schwierigkeit darin zu unterscheiden, ob an den bestehenden String auf dem LCD-Display angehängt , oder ob das Display neu beschrieben werden soll.

Dies wird durch folgenden kleinen Code-Block erreicht:

```
static int taihu_lcd_open( struct inode *dev_file, struct file *f_instance )
{
```

```
    if(!(f_instance->f_flags&O_APPEND)) {               (1)
        iowrite8(CMD_CLEAR_DISPLAY,(void*)cmd_mmap);    (2)
        udelay(2000);
        g_addr=CMD_SET_HOME;                            (3)
    }
    return 0;
}
```

1. Zuerst wird überprüft ob am aktuellen Filedescriptor das O_APPEND Flag gesetzt ist
2. Falls dieses nicht gesetzt ist, wird das ClearDisplay Kommando (0x01) in das Befehlsregister des Displays geschrieben
3. Und anschließend wird nach kurzer Wartezeit der interne Zähler wieder auf den ersten Buchstaben der ersten Zeile des Displays gesetzt.

Durch diesen kurzen Code-Block wird nun zwischen Schreiben und Anhängen unterschieden - zusammen mit der Write Methode war es ab diesem Zeitpunkt möglich z.B. mit dem einfachen `echo` Kommando auf das Display zu schreiben:

```
echo -n 'Hello' > /dev/lcd
echo -n 'World!' >> /dev/lcd
```

Das Character Device `/dev/lcd` musste vorher natürlich mittels `mknod /dev/lcd c 253 0` erstellt werden und der Treiber geladen sein.

Die Write Methode (`taihu_lcd_write`) ist nur unwesentlich komplexer als die Open Methode und deshalb wird an dieser Stelle nicht genauer darauf eingegangen.

### 2.3.4. Integration in sysfs / Schnittstelle für Rohdaten

Die nächste Teilaufgabe bestand nun darin, den Treiber in das sysfs einzubinden und somit eine komfortable Schnittstelle zu den Internas des Treibers zu anzubieten.

Das unter `/sys` gemountete sysfs ist ein virtuelles Dateisystem, welches diverse Informationen des Kernels in den Userspace exportiert - neben Informationen über Interrupts, Modulen und Geräten gibt es im sysfs auch diverse Dateien, mit denen sich das Verhalten des Kernels beeinflussen und einstellen lässt.

Alle Dateien im sysfs sind in Wirklichkeit keine echten Dateien, sondern vielmehr nur ein Abbild von den Strukturen innerhalb des Kernels in

Dateiform - dabei besitzt jedes Kernel-Objekt (`kobject`) einen oder mehrere Einträge im sysfs.

Das sysfs wird insbesondere auch dazu verwendet um dem erfahrenen Anwender eine einfache Schnittstelle zum Konfigurieren von Geräten anzubieten - dies wurde früher durch Einträge im `/proc` Dateisystem realisiert, was jedoch heute als veraltet gilt.

Zu jeder dieser Konfigurationsdatei werden in der Regel im Treiber je eine Lese und eine Schreibfunktion hinterlegt - in unserem Beispiel gibt es für die Ansteuerung der Hintergrundbeleuchtung eine `get_backlight` und eine `set_backlight` Methode.

Liest man die `backlight` Datei unseres Treibers wird die `get_backlight` Funktion aufgerufen, beim Schreiben entsprechend die `set_backlight` Funktion.

Es gibt theoretisch mehrere Möglichkeiten seine Schnittstellen in das sysfs zu exportieren - im `Linux Device Drivers` Buch von Jonathan Corbet, Alessandro Rubini, und Greg Kroah-Hartman sind im Kapitel 14 folgende Möglichkeiten beschrieben:

- Manuelles Registrieren der `kobjects`
- Simple Class Funktionen - ein einfaches Interface um Dateien unter `/sys/class/Klassenname` einzuhängen
- Class Funktionen - ein etwas komplizierteres Interface um Dateien unter `/sys/class/Klassenname` einzuhängen
- Registrieren eines Devices

Während die erste Methode hauptsächlich zur Erläuterung der Internas dient, erschien insbesondere das Class-Simple Interface als die passende Methode für meinen Treiber - dabei gab es jedoch nur ein Problem

### 2.3.5. Problem: API-Änderungen im Kernel

Leider bezieht sich das Linux Device Drivers Buch auf eine sehr frühe Kernel Version der 2.6er Serie (genauer gesagt 2.6.8) und ist somit im Vergleich zu der eingesetzten Kernel Version des Taihu-Boards (2.6.13) bereits veraltet.

Eine Besonderheit des Linux Kernels ist dessen relativ kurzer Release-Zyklus, ca. alle 2 Monate erscheint eine neue Version mit neuen Features. Aber neben neuen Features werden auch alte Funktionen und Konzepte überarbeitet - bereitet ein Feature mehr Probleme, als dass es löst oder hat es sich in der Praxis als untauglich erwiesen, wird es entweder überarbeitet oder es wird nach einiger Zeit komplett entfernt und durch ein besseres ersetzt.

Genau dies geschah auch mit dem oben erwähnten `Class Simple` Interface - dieses wurde in das normale `Class` Interface integriert, somit wurde die Funktionalität erhalten, aber leider auch die Kompatibilität zur alten API

gebrochen - siehe http://thread.gmane.org/gmane.linux.kernel/311695

Ab diesem Zeitpunkt stand das Buch mir nicht mehr als Referenz zur Verfügung und ich musste selbst herausfinden, wie das neue `Class` Interface funktionierte. Glücklicherweise fand sich auf einer Mailingliste die Lösung - die Funktionen haben lediglich andere Namen erhalten - somit konnte ich meinen Code relativ schnell auf die neue API anpassen.

Jedoch stellte sich heraus, dass auch das Class Interface zwischenzeitlich als `deprecated` eingestuft wurde und seit Kernelversion 2.6.19-rc1 gar nicht mehr zur Verfügung steht - der Hintergrund hiervon ist, dass das sysfs durch die vielen Dateien, die meist an willkürlichen Stellen eingepflegt wurden, drohte zu unübersichtlich und damit nutzlos zu werden.

Also wurden im nächsten Schritt die Dateien an die Device Struktur des LCD-Display-Treibers gehängt:

```
DEVICE_ATTR(hex_cmd,S_IWUSR,NULL,store_hex_cmd);
DEVICE_ATTR(hex_data,S_IWUSR,NULL,store_hex_data);
//...
lcd_class = class_create (THIS_MODULE,"lcd");
lcd_device = device_create(lcd_class,NULL,dev,NULL,"lcd0");
status = device_create_file(lcd_device,&dev_attr_hex_cmd);
status =device_create_file(lcd_device,&dev_attr_hex_data);
```

Aber da auch dieses Interface einigen Änderungen unterlag - so heißen die Strukturen und Methoden anders als in früheren Kernelversionen - wurde auch dieser Ansatz verworfen, obwohl er unter 2.6.13 ohne Probleme lief und durch Anpassung der Namen auch unter neueren Kerneln funktioniert hätte.

Der Grund warum dieser Code von mir verworfen wurde war, dass ich auf `/sys/class/misc` gestoßen bin - einem Platz im sysfs für *miscellaneous devices* und das LCD-Display fällt definitiv unter die Kategorie *diverse Geräte*. Der Ansatz, das LCD-Display, unter diesem Teilbaum einzuhängen erschien viel sauberer, als eine ganze LCD-Device-Klasse im sysfs anzulegen, für ein einzelnes Gerät!

Außerdem bringt das `misc` Interface noch diverse weitere Erleichterungen mitsich - so wird das Device mit einer eigenen `Minor` Nummer und der Major von `misc` in das System eingehängt - somit verschwenden wir einerseits keine wertvolle Majornummer und andererseits können wir den kompletten Code zur alloziierung der Majornummer einsparen.

Aber leider wurde auch dieses Interface einer Änderung unterzogen - durch das klare und gekapselte Design halten sich die Änderungen aber in Grenzen und könnten theoretisch durch einfache `#ifdef` Anweisungen realisiert werden.

Die passende Major und Minor Nummer können wir theoretisch aus der

Datei `/sys/class/misc/lcds/dev` lesen und anschließend unser char device mit `mknod` erstellen

### 2.3.6. sysfs und udev

Jedoch ist dies Dank sysfs nicht mehr nötig - durch den Hotplug Mechanismus wird für jedes neue Gerät ein Script aufgerufen - in unserem Fall `mdev`, der kleinen Bruder von `udev`

`mdev` sucht dann für unser Device im sysfs den passenden Eintrag mit dem Namen `dev` - und findet diesen unter dem oben erwähntem Pfad `/sys/class/misc/lcds/dev`

Anschließend legt mdev für uns das passende Device an - wir brauchen uns also um nichts mehr händisch kümmern.

Die Unterstützung für mdev ist im ursprünglichen Root-Filesystem des Taihu-Boards nicht enthalten - deshalb wurde die ursprüngliche Busybox durch eine neue ersetzt, welche mdev beinhaltete.

Anschließend muss man noch folgende Schritte unternehmen, damit der Hotplug-Mechanismus funktioniert:

1. mount -t tmpfs mdev /dev
2. mkdir /dev/pts
3. mount -t devpts devpts /dev/pts
4. mount -t sysfs sysfs /sys
5. echo /bin/mdev > /proc/sys/kernel/hotplug
6. mdev -s

Dies kann jedoch durch die Anpassung des Initscriptes automatisiert werden.

### 2.3.7. Ergebnis

An dieser Stelle möchte ich abschließend den fertigen LCD-Treiber präsentieren und dabei die ein oder andere Code-Zeile etwas näher beleuchten.

```
00001: /*
00002:
00003:    Copyright 2008 Peter Huewe <peterhuewe (at) gmx.de>
00004:
```

```
00005:    This program is free software: you can redistribute it and/or modify
00006:    it under the terms of the GNU General Public License as published by
00007:    the Free Software Foundation, either version 3 of the License, or
00008:    (at your option) any later version.
00009:    This program is distributed in the hope that it will be useful,
00010:    but WITHOUT ANY WARRANTY; without even the implied warranty of
00011:    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
00012:    GNU General Public License for more details.
00013:    You should have received a copy of the GNU General Public License
00014:    along with this program.  If not, see <http://www.gnu.org/licenses/>.
00015:
00016:
00017: */
00018:
00019: #include <linux/module.h>
00020: #include <linux/moduleparam.h>
00021: #include <linux/init.h>
00022:
00023: #include <linux/kernel.h>   /* printk() */
00024: #include <linux/slab.h>   /* kmalloc() */
00025: #include <linux/fs.h>       /* everything... */
00026: #include <linux/errno.h>    /* error codes */
00027: #include <linux/types.h>    /* size_t */
00028: #include <linux/mm.h>
00029: #include <linux/kdev_t.h>
00030: #include <asm/page.h>
00031: #include <linux/cdev.h>
00032:
00033: #include <linux/device.h>
00034: #include <linux/ioport.h>
00035: #include <linux/delay.h>
00036: #include <asm/io.h>
00037: #include <asm/uaccess.h>
00038: #include <linux/errno.h>
00039: #include <linux/stat.h> //Modes
00040: #include <linux/miscdevice.h>
00041:
00042: /*(1)*/
00043: #define LCD_BCKL_ADDR  0x50100001
00044: #define LCD_CMD_ADDR  0x50100002
00045: #define LCD_DATA_ADDR 0x50100003
00046: #define CMD_CLEAR_DISPLAY 0x01
00047: #define CMD_SET_HOME 0x80
00048:
00049:
00050:
00051: /*(2)*/
00052: static volatile void* data_mmap;
00053: static volatile void* cmd_mmap;
00054: static volatile void* bckl_mmap;
00055: static unsigned char g_addr;
00056:
00057: static int taihu_lcd_open( struct inode *dev_file, struct file *f_instance );
00058: ssize_t taihu_lcd_write(struct file *filp,const char __user *buf, size_t count, loff_t *f_pos
00059: static int __init taihu_lcd_init(void);
00060: static void __exit taihu_lcd_cleanup(void);
00061:
```

```
00062:
00063: static int taihu_lcd_open( struct inode *dev_file, struct file *f_instance )
00064: {
00065:     if(!(f_instance->f_flags&O_APPEND)) { // If we do not append to device we flush it
00066:         iowrite8(CMD_CLEAR_DISPLAY,(void*)cmd_mmap); //flush and return home
00067:         udelay(2000);
00068:         g_addr=CMD_SET_HOME; //set cursor to first line, first character
00069:     }
00070:     return 0;
00071: }
00072:
00073:
00074: /*(3)*/
00075: ssize_t taihu_lcd_write(struct file *filp,const char __user *buf, size_t count, loff_t *f_pos
00076: {
00077:     size_t i=0;
00078:     unsigned char *ks_buf=kmalloc(count+1,GFP_KERNEL);//+1 for nullbyte
00079:     (*f_pos) += count; // TODO check how many bytes writen.
00080:     if(ks_buf==NULL)
00081:     {
00082:         printk (KERN_EMERG "Kmalloc failed!\n");
00083:     }
00084:     else
00085:     {
00086:         if(!copy_from_user(ks_buf,buf,count))
00087:         {
00088:             for (i=0; i< count;i++)
00089:             {
00090:                 iowrite8(ks_buf[i],(void*) data_mmap);
00091:                 udelay(2000);
00092:                 g_addr++;
00093:                 if(g_addr & 0x10) //end of lineclass_ reached - change to other line (4)
00094:                 {
00095:                     g_addr^=0x40; // Toggle Second line
00096:                     g_addr&=0xC0; // Reset cursor to first char of line
00097:                     iowrite8(g_addr,(void*) cmd_mmap);
00098:                     udelay(2000);
00099:                 }
00100:             }
00101:         }
00102:         else
00103:         {
00104:             ks_buf[count]='\0';//Nullbyte for printing
00105:             printk("Copy from user failed! Value %s anzahl %d\n",ks_buf,count);
00106:         }
00107:         kfree(ks_buf);
00108:     }
00109:     return count;
00110: }
00111:
00112:
00113:
00114: /*(5)*/
00115: static struct file_operations taihu_lcd_ops = {
00116:     .owner    = THIS_MODULE,
00117:     .open = taihu_lcd_open,
00118:     .write    = taihu_lcd_write
```

```
00119: };
00120:
00121:
00122:
00123:
00124: /*(6)*/
00125: static ssize_t store_hex_cmd (struct class_device *dev, const char *buffer,size_t size)
00126: {
00127:     char new = simple_strtol (buffer,NULL,16);
00128:     iowrite8(new, (void*)cmd_mmap);
00129:     return 4;
00130: }
00131: static ssize_t store_hex_data (struct class_device *dev,const char *buffer,size_t size)
00132: {
00133:     char new = simple_strtol (buffer,NULL,16);
00134:     iowrite8(new, (void*)data_mmap);
00135:     return 4;
00136: }
00137: static ssize_t store_cmd (struct class_device *dev, const char *buffer,size_t size)
00138: {
00139:     iowrite8(buffer[0],(void*) cmd_mmap);
00140:     return 1;
00141: }
00142:
00143: static ssize_t store_data (struct class_device *dev, const char *buffer,size_t size)
00144: {
00145:     iowrite8(buffer[0],(void*) data_mmap);
00146:     return 1;
00147: }
00148:
00149:
00150: static ssize_t get_backlight (struct class_device *dev,char *buffer)
00151: {
00152:     ssize_t len =0 ;
00153:     char backlight = ioread8((void *)bckl_mmap);
00154:     backlight >>= 1;
00155:     backlight &=0x01;
00156:     len = snprintf(buffer,3, "%d\n",backlight);
00157:     return len;
00158: }
00159: static ssize_t set_backlight (struct class_device *dev,  const char *buffer,size_t size)
00160: {
00161:     char new;
00162:     char backlight = ioread8((void *)bckl_mmap);
00163:     new = simple_strtol (buffer,NULL,2);
00164:     if(new == 1)
00165:     {
00166:         backlight |= 0x02;
00167:     }
00168:     else if (new == 0) {
00169:         backlight &= ~(0x02);
00170:     }
00171:     else { // Error
00172:         return -EINVAL;
00173:     }
00174:     iowrite8(backlight,(void *)bckl_mmap);
00175:     return size;
```

```
00176: }
00177:
00178:
00179: /*(7)*/
00180: CLASS_DEVICE_ATTR(hex_cmd,S_IWUSR,NULL,store_hex_cmd);
00181: CLASS_DEVICE_ATTR(hex_data,S_IWUSR,NULL,store_hex_data);
00182: CLASS_DEVICE_ATTR(cmd,S_IWUSR,NULL,store_cmd);
00183: CLASS_DEVICE_ATTR(data,S_IWUSR,NULL,store_data);
00184: CLASS_DEVICE_ATTR(backlight,S_IRUGO|S_IWUSR,get_backlight,set_backlight);
00185:
00186: /*(8)*/
00187: static struct miscdevice taihu_miscdev = {
00188:     255, // dynamic minor please :)
00189:     "lcds",
00190:     &taihu_lcd_ops
00191: };
00192:
00193:
00194:
00195: static int __init taihu_lcd_init(void) {
00196:     struct resource *res_cmd_data;
00197:     struct resource *res_lcd_data;
00198:     struct resource *res_lcd_bckl;
00199:     int status=0;
00200:
00201: /*(9)*/
00202:     if((res_cmd_data = request_mem_region(LCD_CMD_ADDR,1L,"lcdcmd"))==NULL)
00203:     {
00204:         printk(KERN_ERR "An error occured while requesting mem_region for lcd_cmd_addr\n");
00205:         goto err_0; /*(10)*/
00206:     }
00207:     if((res_lcd_data = request_mem_region(LCD_DATA_ADDR,1L,"lcddata"))==NULL)
00208:     {
00209:         printk(KERN_ERR "An error occured while requesting mem_region for lcd_data_addr\n");
00210:         goto err_1;
00211:     }
00212:     if((res_lcd_bckl = request_mem_region(LCD_BCKL_ADDR, 1L,"lcdbckl"))==NULL)
00213:     {
00214:         printk(KERN_ERR "An error occured while requesting mem_region for lcd_bckl_addr\n");
00215:         goto err_2;
00216:     }
00217:
00218:
00219: /*(11)*/
00220:     cmd_mmap=ioremap((unsigned long)LCD_CMD_ADDR,1);
00221:     data_mmap=ioremap((unsigned long)LCD_DATA_ADDR,1);
00222:     bckl_mmap=ioremap((unsigned long)LCD_BCKL_ADDR,1);
00223:     iowrite8(CMD_CLEAR_DISPLAY,(void*)cmd_mmap);
00224:     g_addr=CMD_SET_HOME;
00225:     udelay(2000);
00226:
00227: /*(12)*/
00228:     misc_register(&taihu_miscdev);
00229:     status=class_device_create_file(taihu_miscdev.class,&class_device_attr_hex_cmd);
00230:     status=class_device_create_file(taihu_miscdev.class,&class_device_attr_hex_data);
00231:     status=class_device_create_file(taihu_miscdev.class,&class_device_attr_data);
00232:     status=class_device_create_file(taihu_miscdev.class,&class_device_attr_cmd);
```

```
00233:     status=class_device_create_file(taihu_miscdev.class,&class_device_attr_backlight);
00234:     return 0;
00235:
00236:
00237: /*(13)*/
00238:     //err_3:
00239:     printk(KERN_DEBUG "cleanup - releasing mem_region for lcd_cmd_addr\n ");
00240:     release_mem_region(LCD_BCKL_ADDR,1L);
00241: err_2:
00242:     printk(KERN_DEBUG "cleanup - releasing mem_region for lcd_cmd_addr\n ");
00243:     release_mem_region(LCD_CMD_ADDR,1L);
00244: err_1:
00245:     printk(KERN_DEBUG "cleanup - releasing mem_region for lcd_data_addr\n ");
00246:     release_mem_region(LCD_DATA_ADDR,1L);
00247: err_0:
00248:     printk(KERN_DEBUG "Nothing left to clean up - Bailing out\n ");
00249:     return status;
00250: }
00251:
00252: static void __exit taihu_lcd_cleanup(void){
00253: /*(14)*/
00254:     class_device_remove_file(taihu_miscdev.class,&class_device_attr_backlight);
00255:     class_device_remove_file(taihu_miscdev.class,&class_device_attr_cmd);
00256:     class_device_remove_file(taihu_miscdev.class,&class_device_attr_data);
00257:     class_device_remove_file(taihu_miscdev.class,&class_device_attr_hex_data);
00258:     class_device_remove_file(taihu_miscdev.class,&class_device_attr_hex_cmd);
00259:     misc_deregister( &taihu_miscdev );
00260:     release_mem_region(LCD_BCKL_ADDR,1L);
00261:     release_mem_region(LCD_DATA_ADDR,1L);
00262:     release_mem_region(LCD_CMD_ADDR,1L);
00263: }
00264:
00265:
00266: MODULE_AUTHOR("Peter Huewe");
00267: MODULE_LICENSE("GPL");
00268: module_init(taihu_lcd_init);
00269: module_exit(taihu_lcd_cleanup);
```

1. Durch die Verwendung von defines für die verwendeten Hexwerte wird der Code sehr viel lesbarer und besser wartbar.
2. Als nächstes folgen die Funktionsprototypen und globalen Variablen.
3. Die vorhin bereits erwähnte `lcd_write` Funktion - hierbei muss strengstens darauf geachtet werden, dass man nicht auf dem **buf** direkt arbeitet, da es sich hierbei um eine Adresse aus dem Userspace handelt.
4. Hier wurde noch eine kleine Logik eingebaut, damit der Text auf dem Display automatisch umbricht.
5. Die file_operations Struktur ist in (fast) jedem Treiber vorhanden.
6. Darauf folgen die Funktionen, die bei Schreib-/Lesezugriffen auf die sysfs Dateien des Treibers aufgerufen werden

7. Anschließend werden diese Funktionen zusammen mit einem Namen durch Compiler Macros **zur Kompilierzeit** in **class_device_attributes** umgewandelt. Alle Dateien sind nur von Root beschreibar, lediglich *backlight* kann von jedermann auch gelesen werden.
8. Unsere Struktur mit der sich unser Treiber beim `misc` Subsystem registriert.
9. Da die Register des LCD memory mapped sind, müssen diese Bereiche für uns reserviert werden - somit soll kein anderer Treiber versehentlich auf diese Bereiche schreiben können.
10. Goto ist bei der Treiberentwicklung ein immernoch häufig eingesetztes Konstrukt, welches oftmals zur effizienten Fehlerbehandlung verwendet wird.
11. Im nächsten Schritt müssen die Adressen der Register in Adressen des Kernelspace umgewandelt werden.
12. Danach registriert sich der Treiber beim misc Subsystem und legt seine Dateien im sysfs an.
13. Traten Fehler auf, müssen die Schritte in umgekehrter Reihenfolge rückgängig gemacht werden
14. Dasselbe gilt auch beim Entladen des Moduls,

### 2.3.8. Schnittstellen

Der Treiber exportiert in den Userspace folgende Schnittstellen:

| Pfad | Beschreibung |
|---|---|
| /dev/lcds | Character Device - kann mittels normalen Dateizugriffen beschrieben werden |
| /sys/class/misc/lcds/dev | Diese Datei beinhaltet die Major und Minor Nummer für mdev |
| /sys/class/misc/lcds/data | Schnittstelle für Rohdaten als Integers |
| /sys/class/misc/lcds /hex_data | Schnittstelle für Rohdaten als Hex-Strings! (z.B. "0x65") |
| /sys/class/misc/lcds/cmd | Schnittstelle für Befehle als Integers |
| /sys/class/misc/lcds /hex_cmd | Schnittstelle für Befehle als Hex-Strings! (z.B "0x01") |
| /sys/class/misc/lcds /backlight | Schnittstelle zum Steuern der Hintergrundbeleuchtung (1 = an / 0 = aus) |

Die beiden `hex_XXX` Schnittstellen wurden explizit für das Shellscripting implementiert, da ein `echo 0x10` einen String übergibt und keine Zahl!

## 2.4. Fazit

Die Entwicklung des Treibers und die Arbeit allgemein mit dem Taihu-Board hat sehr viel Spaß gemacht und ich habe vorallem im Bereich sysfs viele neue Erfahrungen gesammelt. Außerdem hat die Entwicklung dieses Treibers wiedereinmal veranschaulicht, dass man unbedingt die verschiedenen Entwicklungen im Linux-Kernel im Auge behalten muss um nicht aufgrund fehlender oder geändertet APIs ins Hintertreffen zu geraten.

Die verschiedenen API Änderungen waren anfangs sehr frustrierend, da man im erstem Moment relativ ratlos ist warum der Code nicht funktioniert, bzw. wie man ihn **richtig** abändert. Meines Erachtens ist wäre eine aktualisierte Auflage von `Linux Device Drivers` für viele Neulinge in Sachen Linux-Treiber eine sehr große Hilfe - denn derzeit kann man sich nur mit viel Mühe durch das git repository hangeln um herauszufinden, wie andere Projekte die Probleme gelöst haben.

Sehr positiv hervorzuheben war das Wiki von denx.de - dort wurden viele Fragen rund um U-boot sehr gut beantwortet und erklärt und auch das ELDK viel sehr positiv auf, da die Toolchain out-of-the-box funktionierte und während der ganzen Entwicklung keinerlei Probleme mit dieser auftauchten.

Abschließend kann man sagen, dass der Treiber zwar mit seinen knapp 270 Zeilen nicht besonders umfangreich ausgefallen ist - aber dennoch stecken in diesen Zeilen unzählige Stunden Arbeit, die sich nicht mit Lines of Code pro Stunde messen lassen - und das obwohl ich bereits Vorkenntnisse in diesem Bereich besaß!

Dennoch eine sehr schöne und spannende Aufgabe!


# 3. NGW100

Als zweiten Teil des Praktikums habe ich ein wenig mit dem NGW100 Board

von Atmel experimentiert - dieses Board konnten wir im Rahmen des Atmel Studentenprogramm relativ günstig beziehen und eignet sich deshalb besonders gut für Versuche im Bereich Embedded Linux.

Das Board bietet mit seiner 150 MHz getakteten CPU, den 2 Ethernetanschlüssen, sowie dem SD Kartenleser und dem relativ kleinen Formfaktor eine ideale Einstiegsplattform und durch die vielzähligen GPIO Erweiterungen sehr viele Möglichkeit das Board z.B. um ein LCD-Display, PS/2 Anschluss und Sound zu erweitern.

## 3.1. Idee

Meine eigentliche Idee für das Board war die Einrichtung eines kleinen Rechners mit TFT Display, mit dem ich im Internet über ICQ, Jabber, IRC chatten und evtl surfen kann. Außerdem wäre es sehr schön wenn man dabei noch etwas Musik hören könnte, welche sich auf einem anderen Rechner im Netzwerk befindet.

Der NGW100 eignet sich aufgrund seiner geringen Leistungsaufnahme von ca 1,3 Watt auch ohne weiteres für einen allways-on Betrieb, ohne dass dabei die Stromrechnung in die Höhe schießt - im Gegenteil, die meiste Zeit könnte der kleine Rechner meinen 150W verschwendenden Desktoprechner mühelos ersetzen.

Wenn man außerdem eine geeignete mobile Stromversorgung beschaffen könnte, würde sich der NGW100 auch als Laptopersatz eignen - zusammen mit einem kleinen Solar-Panel ergäbe sich hierbei sicherlich eine beträchtliche Akku-Laufzeit!

## 3.2. Manuelles Erstellen der Toolchain

Um einen tieferen Einblick in das Cross Compilieren zu bekommen habe ich mich entschlossen, meine Toolchain selbst aufzusetzen damit diese (nur) die Features aufweist, die ich brauche.

Folglich habe ich mir die einzelnen Tools direkt von Atmel heruntergeladen und bin anschließend nach der Anleitung im AVRfreaks Wiki - Linux Toolchain HowTo vorgegangen.

Sämtliche Schritte verliefen nur mit kleinen Problemen, lediglich beim finalen GCC gab es Schwierigkeiten mit den pthreads - hierbei brach der Kompiliervorgang immer wieder ab - nachdem jedoch keine Lösung in Sicht war und ich bereits einige Zeit in das manuelle Erstellen Toolchain gesteckt hatte, wurde dieser Versuch an dieser Stelle abgebrochen.

## 3.3. Buildroot

Buildroot ist Softwarepaket, welches im Prinzip nur auf einem Haufen Makefiles basiert, mit dem man auf relativ einfache Art und Weise für ein bestimmtes Target eine Cross-Toolchain erstellen kann - mit welcher anschließend automatisch ein Root-Filesystem für das Target generiert wird.

Dabei setzt Buildroot auf das vom Linux-Kernel bekannte `Kconfig` Buildsystem - so gibt es auch hier die bekannten Optionen `menuconfig` und `boardname_defconfig`. Mittels `menuconfig` kann der Benutzer von Buildroot relativ einfach auswählen welche Pakete und Features für das Targetsystem gewünscht werden.

Soweit die Theorie - in der Praxis verhält sich Buildroot leider nur für die jeweilige Board Default Config stabil - fügt man Pakete hinzu oder entfernt diverse Features wird i.d.R. **nicht mehr** überprüft ob die Toolchain noch zu den gewünschten Paketen passt! - mehr dazu im Abschnitt Kritik.

Atmel selbst bietet eine angepasste Version von Buildroot zum Download.

### 3.3.1. Eigene Pakete

#### Midnight Commander

Das erste Programm, welches ich für das Board portieren wollte, war der berühmte Filemanager `midnight commander` oder auch kurz `mc` genannt. Jedoch stellte sich die Portierung dieses Paketes als sehr sehr zeitintensiv heraus:

Einerseits ist mc ein sehr komplexes Programm, mit sehr vielen optionalen Erweiterungen und damit auch sehr vielen Abhängkeiten - diese manuell sauber aufzulösen ist nicht trivial und erfordert viel Zeit und Geduld.

Andererseits erfolgt dass Erstellen von mc in mehreren Phasen, zuerst wird der Quellcode kompiliert, sodass anschließend die fertigen Binaries zur Verfügung stehen. Unter diesen Binaries befinden sich jedoch auch Tools zum Erstellen der Dokumentation, welche von jedem sub-makefile unbedingt gebaut werden will. Genau hierbei liegt dann das Problem - die Tools werden mittels Crosscompiler für AVR32 kompiliert, und anschließend müssen die Tools aber auch auf dem Hostrechner ausgeführt werden - was natürlich nicht funktioniert.

Nach mehreren Tagen des makefile Patchens und immer neuen Versuchen wurde die Portierung von mc abgebrochen und nach einer Alternative umgeschaut.

#### Clex

Dabei stieß ich auf clex - dieser ncurses basierte Filemanager ist mit einer gepackten Downloadgröße von 214 Kb relativ leichtgewichtig im Vergleich zum 3,8 Mb schweren Midnight Commander!

Um Clex in Buildroot zu integrieren habe ich eine sogenanntes .mk-Rezept geschrieben - dieses beschreibt wie Buildroot das Paket entpacken, konfigurieren und compilieren muss,

Dieses Rezept sowie die Config.in Datei, welche für das KConfig-Buildsystem benötigt wird und ein Patch welcher dafür sorgt dass die Rücktaste richtig funktioniert finden sich im Ordner scr/clex/

Diesen Ordner muss man nur in das Verzeichnis `package` unterhalb von Buildroot kopieren - fügt man in diesem Ordner der Datei `Config.in` noch folgenden Eintrag hinzu `source "package/sdl_image/Config.in"` steht das Clex Paket nun auch innerhalb von `menuconfig` zur Verfügung und kann somit sehr einfach dem Root-Filesystem hinzugefügt werden.

Der Clex-Manager wurde natürlich auch als Featurerequest an Buildroot gemeldet - wobei er natürlich bisher nicht beachtet wurde - bzw. beim Upgrade auf einen neuen Bugtracker hat Buildroot leider alle bisherigen Bugreports gelöscht!

### Pidgin/Finch

Das wichtigeste Paket für mein Vorhaben war jedoch ein ICQ/Jabber Client der mit einem einfachen ncurses Interface auskommt - denn eine große Grafik-Library hat auf dem kleinen Flash nur bedingt platz.

Es wurden sämtliche mit bekannten Clients ausprobiert, darunter `CenterIM`, `climm` und `licq` jedoch lies sich keines dieser Pakete zufriedenstellend compilieren - insbesondere bei CenterIM gab es massive Probleme mit dem Auffinden von ncurses.

Also wurde im nächsten Schritt die Konsolenvariante von `Pidgin`, einem Fork von `Gaim`, ausprobiert, die auf den Namen `finch` hört. Sowohl Finch als auch Pidgin basieren auf der `libpurple` diese Library entwickelt sich derzeit zum Quasi-Standard beim Zugriff auf Instant Messenging Dienste unter Linux - so steigt z.B. auch das CenterIM Projekt derzeit auf diese Library um!

Mehr Details zu Pidgin, Finch und Libpurple finden sich auf http://pidgin.im

Die Dateien um Finch mit Buildroot zu kompilieren finden sich unter scr/Pidgin/

Da es sich bei Finch um ein Programm handelt, dass mit den autotools konfiguriert werden kann, sieht die .mk datei ein klein wenig anders aus:

```
#############################################################
#
# pidgin
#
#############################################################
PIDGIN_VERSION = 2.5.2
PIDGIN_SOURCE = pidgin-$(PIDGIN_VERSION).tar.bz2
PIDGIN_SITE = http://puzzle.dl.sourceforge.net/sourceforge/pidgin/
PIDGIN_DIR = $(BUILD_DIR)/pidgin-$(PIDGIN_VERSION)
PIDGIN_AUTORECONF = NO
PIDGIN_CONF_ENV = ac_cv_path_pythonpath=''
PIDGIN_CONF_OPT=--target=$(GNU_TARGET_NAME) \
                --host=$(GNU_TARGET_NAME) \
```

```
                --build=$(GNU_HOST_NAME) \
                --prefix=/usr \
                --sysconfdir=/etc \
                                --disable-gtkui \
                --enable-consoleui \
                --disable-screensaver \
                --disable-sm \
                --disable-gtkspell \
                --disable-gestures   \
                --disable-schemas-install \
                --disable-gstreamer \
                --disable-gstreamer  \
                --disable-meanwhile \
                --disable-avahi \
                --disable-fortify \
                --disable-dbus  \
                --disable-nm  \
                --disable-mono \
                --disable-perl \
                --enable-gnutls=no \
                --enable-nss=no \
                --disable-tcl \
                --disable-tk \
                --disable-pixmaps-install \
                --enable-nls \
                --disable-doxygen \
                --without-x \
                --with-static-prpls=oscar,jabber,irc \
                --without-python \
                --disable-plugins \
                --disable-debug \
                $(DISABLE_NLS) \
                $(DISABLE_LARGEFILE) \

PIDGIN_DEPENDENCIES = uclibc gettext libintl libglib2
$(eval $(call AUTOTARGETS,package,pidgin))
```

Hierbei sieht man, dass man einige Optionen von Finch abschalten muss um sich nicht allzuviele Abhängigkeiten einzufangen.

Besonders erwähnenswert ist noch die Zeile `PIDGIN_CONF_ENV = ac_cv_path_pythonpath=''` - hierbei handelt es sich um einen kleinen Trick um den Autotools vorzutäuschen, dass auf dem Hostsystem kein Python installiert ist - setzt man diese Option nicht, wird das finch Binary gegen python gelinkt, was natürlich aufgrund der unterschiedlichen Architekturen fehlschlägt.

Leider ist dieses Rezept noch nicht ganz ausgereift - auf dem Targetsystem findet sich zwar etwas unter `/usr/bin/finch`, jedoch handelt es sich hierbei lediglich um ein Wrapperscript, welches die verschiedenen Libraries beim ersten Aufruf initialisieren soll - das eigentliche Binary findet sich unter `/usr/bin/avr32-linux-finch`.

Jedoch ist dieses Binary nicht gestripped worden, da Buildroot eine Binary mit dem Namen `finch` erwartet - somit ist finch in diesem Zustand noch viel

zu groß und muss von Hand runtergestripped werden.

Außerdem funktioniert das Binary nur mit passenden Umgebungsvariablen, sodass man sich am besten ein shellscript dafür anlegt mit folgendem Inhalt: `TERM=xterm LANG=en_US.UTF-8 LC_LANG=en_US.UTF-8 /usr/bin/avr32-linux-finch` damit lässt sich finch ohne größere Probleme im Jabber und IRC Netzwerk verwenden!

Leider hat das setzen der Sprache auf `en_US.UTF-8` einen entscheidenden Nachteil - das ICQ-Protokoll versteht nur Nachrichten im Latin1 Encoding! Somit kann `finch` auf meinem NGW100 lediglich Nachrichten von ICQ empfangen, jedoch keine Nachrichten senden.

Auch wenn der Buildprozess und meine `finch` Binary noch diverse Schwächen aufweisen, ist durch das obige Paket in jedemfall schonmal der Grundstein dafür gelegt den NGW100 chatfähig zu machen.

### 3.3.2. Kritik

Die Arbeit mit Buildroot war oftmals sehr frustrierend, denn es traten oftmals Fehler auf die den wahren Grund des Problems nicht direkt anzeigten und man viel Zeit mit der Suche an der falschen Stelle verbracht hat.

#### Makereihenfolge beachten

Eine der kleinen aber nervigen Baustellen von Buildroot ist, dass man sich beim ersten Erstellen **genau** an folgende Reihenfolge halten muss:

- make atngw100_defconfig
- make // abbrechen wenn Linux-Kernel heruntergeladen wird
- evtl alte Config reinkopieren
- make menuconfig // Konfigurieren
- make

hält man sich nicht an diese Reihenfolge, bekommt sporadisch Fehlermeldungen wie z.B.:

```
ln: `/home/avr32/buildroot/build_avr32/staging_dir/usr/.': cannot overwrite directory
make: *** [/home/avr32/buildroot/build_avr32/staging_dir] Error 1
```

Der Ordner usr existiert jedoch gar nicht! - entpackt man Buildroot neu und macht dasselbe in o.g. Reihenfolge funktioniert es.

#### Nach Veränderungen an der Toolchain Config wird diese nicht neu gebaut

Ein besonders nerviger Bug ist, dass Buildroot eine einmal erstelle Toolchain nicht neu erstellt, selbst wenn man an der Config der Toolchain

etwas geändert hat. Dies ist besonders frustrierend wenn man nachträglich ein Tool zu seinem Root-Filesystem hinzufügen will, welches eine Änderung an der Toolchain benötigt - hierbei muss man die Toolchain, sowie alle Pakete von Hand löschen - da es leider keinen `clean` Befehl gibt, der die Pakete **und** die Toolchain bereinigt.

Die einfachste Methode ist es hierbei die `.config` zu sichern, den `dl` Ordner aus dem Buildroot Verzeichnis per `mv` zu verschieben und anschließend den Buildroot Ordner zu löschen - anschließend entpackt man Buildroot wieder, führt die oben genannten Schritte aus und kann nach ca. 2 Stunden Kompilierzeit sehen, ob die neue Toolchain das Paket bauen kann - wenn nicht heißt es wieder von Vorne anfangen!

Im Laufe der Zeit habe ich weit über 50 mal diesen Schritt ausführen müssen! - meistens nur wegen kleinen Änderungen.

Im Zusammenhang mit dem ersten Bug ist dies natürlich sehr frustrierend!

Da das Löschen des Ordners sehr viel Zeit beansprucht, ist es i.d.R. sinnvoller während der Entwicklung den Ordner nur umzubenennen und anschließend z.B. per Cronjob zu löschen.

### 3.3.3. Buildroot auf Oktopus

Da der Entwicklungszyklus mit Buildroot sich durch die lange Wartezeit von über 2 Stunden pro Durchlauf als sehr ineffizient und unproduktiv herausstellte, wurde versucht, das ganze durch mehr Rechenpower zu beschleunigen - die Lösung des Problems nennt sich `Oktopus` - dabei handelt es sich um einen Server der FH-Augsburg, mit einem 4-Kern Xeon Prozessor als CPU!

Zusammen mit Herrn Prof. Dr. Klüver, wurden für alle Pakete installiert, die für Buildroot notwendig sind - nachdem dies geschehen ist konnte der erste Testlauf starten: Buildroot brauchte für die Default-Konfig nur noch knapp 25 Minuten! - eine dramatische Verbesserung im Vergleich zu meinem Desktoprechner.

Um diesen Vorgang noch weiter zu beschleunigen, wurde einerseits versucht die Anzahl der parallelen Build-Vorgänge zu optimieren, andererseits wurde versucht durch den Einsatz von alternativen zu tar,gzip und bzip2 noch ein kleines Quentchen Performance herauszuquetschen.

Dabei zeigte sich, dass Buildroot auf Oktopus am besten mit der Makeoption `-j5` aufgerufen wird, und dass der Einsatz von z.b. `pbzip2` keine nennenswerten Geschwindigkeitsvorteile erbrachte - in dieser Konfiguration benötigt ein Buildroot-Durchlauf nur noch knapp 20 Minuten!

### 3.3.4. Fazit

Buildroot ist von der Idee her ein sehr schönes tool, das vor allem Einsteigern einen komfortablen Weg bietet um das Root-Filesystem für Ihr Board zu konfigurieren. Jedoch hat Buildroot in der Praxis noch mit diversen Problemen zu kämpfen - dies liegt aber unter anderem auch daran, dass Buildroot eine ganze Palette von Architekturen unterstützt.

## 3.4. Open Embedded

Als Alternative zu Buildroot ist mir unter anderem Open Embedded aufgefallen - Open Embedded bietet Unterstützung für eine ganze Reihe von Target Boards unter anderem auch für den NGW100.

Der große Vorteil von Open Embedded ist, dass man damit sowohl einzelne Pakete erstellen kann, sowie auch ganze Repositories - denn im Gegensatz zu Buildroot steht nicht das Root-Filesystem im Vordergrund sondern einfach zu installierende Pakete - im sogenannten *ipkg* Format.

Hierbei ist vielleicht erwähnenswert, dass man in Buildroot den Paketmanager für das ipkg Format für das Targetboard aktivieren kann.

Das Problem an OE ist jedoch, dass es einem nicht annähernd soviele Freiheiten bietet wie Buildroot - und wenn etwas nicht kompiliert, dann kann man an dieser Stelle nicht viel tun.

Ich persönlich kenne Open Embedded noch von früher als ich meinen Linux-PDA, den Zaurus SL 5600 noch regelmäßig mit neuer Software füttern musste.

Eine Anleitung zu Open Embedded und dem NGW100 findet sich unter: http://montamer.blogspot.com/2008/07/building-linux-for-atngw100-from-oe.html

### 3.4.1. OE auf Oktopus

Um den anderen Teilnehmern im Kurs ein großes Softwarerepository zur Verfügung zu stellen, habe ich als nächstes versucht die große Rechenpower von Oktopus erneut zu nutzen.

Nachdem alle benötigten Abhängigkeiten installiert waren, wurde versucht das erste Paket damit zu erstellen - jedoch scheiterte dies mit *compiler cannot create executables*.

Ein kurzer Blick in die Logfiles vereit, dass die Option `hash-style=gnu` nicht unterstützt wird - eine Recherche im Internet ergab jedoch keine Lösung.

## 3.5. Hardware Modifikationen

Der NGW100 bietet mit seinen Expansion Headers - quasi herausgeführte

GPIO Pins - unzählige Erweiterungsmöglichkeiten - für die Laptop bzw. Chatclient Idee musste der NGW nun noch um ein TFT Display sowie um einen PS2 Anschluss für die Tastatur erweitert werden.

### 3.5.1. LCD Display

In der Application Note AVR32114 beschreibt Atmel relativ genau die einzelnen Schritte, wie man ein LCD-Display an den NGW anschließt.

Nach dieser Anleitung habe ich nun das von Herrn Högl zur Verfügung gestellte 6,4" LCD-Display LB064V02-A1 von LG an den NGW angeschlossen.

Hierfür verwendete ich eine Pfostenstecker für Flachbandkabel, wie man ihn von IDE-Kabeln kennt, und klemmte die einzelnen Adern gemäß der Belegung in den Stecker - aber leider braucht das Display 6 GND Leitungen, während der NGW an diesem Port nur 2 Stück zur Verfügung stellt.

→ An dieser Stelle war kein sauberer Kontakt zwischen den Adern und dem Stecker möglich und sämtliche Tests schlugen aufgrund von Wackelkontakten fehl - es stellte sich jedoch bereits das LCD-typische leichte Pfeifen ein und einen Rand sah man auch bereits.

An dieser Stelle musste eine bessere Lösung gefunden werden.

### 3.5.2. Keyboard

Die zweite Hardwareerweiterung war das Hinzufügen eines PS2 Interfaces an Port J5 - dies ist zwar in einer weiteren Aplication Note von Atmel (AVR32415) genau beschrieben, jedoch liefern die GPIO Pins des NGW nur 3,3 V, während das PS2 Protokoll 5V Pegel verlangt.

Der in der Application Note erwähnte Pegelwandler `PCA9306` wurde mir von Texas Instruments freundlicherweise als Sample zur Verfügung gestellt - jedoch ist dieser so klein, dass er ohne passende Platine unlötbar ist.

Deshalb habe ich nun versucht eine Tastatur zu finden, die auch mit 3,3 V funktioniert - denn im Prinzip wird bei der normalen 5V TTL alles ab ca 2,7 V als logische 1 gewertet.

Die einzige Tastatur die zu funktionieren schien war eine Tastatur von Microsoft - also habe ich mir einen kleinen Adapter gebaut, passend zu den Schaltplänen in der Application Note - jedoch habe ich das ganze durch eine zusätzliche Spannungsquelle ergänzt, denn ohne diese Modifikation bekommt der RAM nicht genügend Strom und das Board bootet nicht.

Bei der externen Spannungsversorgung muss man jedoch aufpassen, dass auch hier lediglich 3,3V anliegen, denn durch die Pull-Up Widerstände wird diese Spannung direkt auf die GPIO Leitungen gelegt - bei 5 Volt hätte dies zur Folge dass der GPIO pin danach kaputt ist!

Deshalb habe ich mir mit einem Widerstand und einem kleinen Potentiometer eine regelbare Spannungsquelle gebaut, die ich relativ exakt

auf 3,3 V einstellen kann - ein vorgeschalteter LM1805 sorgt dabei dafür, dass die Eingangsspannung für meinen Spannungsteiler bei konstant 5V liegt.

> Nochmals der Hinweis: Die GPIO Pins des NGW sind **nicht** 5V tolerant! Legt man 5V an einen der GPIO Pins an ist mindestens dieser Pin endgültig zerstört! - Lieber alles zweimal messen, bevor man die Schaltung an den NGW hängt.

Damit das Keyboard funktioniert, muss im nächsten Schritt der Kernel gepatcht werden und anschließend die Tastatur Unterstützung mittels `menuconfig` aktiviert werden - leider sind jedoch bisher alle Versuche den Kernel zu patchen fehlgeschlagen, denn der neu erstellte Kernel bootet nicht und u-boot startet kurz nach dem Laden neu.

### 3.5.3. Daughterboard

Um die nötigen Spannungen für das Keyboard und den Inverter für die Hintergrundbeleuchtung zu erzeugen, sowie um die passenden Schaltungen für das Keyboard und das LCD-Display zu realisieren, wurde begonnen ein Daughterboard zu löten - dieses wird einfach auf den NGW aufgesteckt und ist somit eine solide Lösung zum Anschluss der Peripherie.

Leider stellte sich das Löten der Lochrasterplatine aufgrund meiner etwas schlechten Lötausrüstung als sehr schwierig und zeitintensiv heraus - hierbei wäre eine geätzte Platine definitiv die einfachere Lösung, jedoch besteht derzeit leider keine Möglichkeit an der FH Platinen zu ätzen - deshalb wurde dieses Vorgehen bis auf weiteres verschoben

### 3.5.4. Batteriebetrieb

Als ich vor kurzem einen "Handy-Notlader" geschenkt bekommen habe, welcher aus einer handelsüblichen 1,5V AA Batterie die benötigten 5V Spannung erzeugt, war meine erste Idee den NGW100 daran anzuschließen - denn der NGW verbrauchte bei einer Messung inklusive Schalternetzteil gerademal 1,3 Watt!

Jedoch fehlte auch diesem Adapter die nötige Leistung um die Stromspitzen beim Initialisieren des RAMs abzufangen - der Strom bricht ein und das Board startet neu.

Dennoch hat mich der Adapter auf eine gute Idee gebracht - da das Board bis zu 12 V toleriert, kann man es auch ganz einfach an einer normalen 9V Block Batterie betreiben - verwendet man statt einer Batterie einen Akku und fügt dem ganzen noch ein kleines Solar-Panel hinzu, könnte man durchaus sehr gute Laufzeiten erreichen.

Die Gesamtlaufzeit an der 9V Batterie wurde jedoch nicht explizit getestet.

### 3.6. Fazit und Ausblick zum NGW100

Der NGW100 von Atmel ist ein sehr schönes Einsteigerboard, welches trotz seines günstigen Preises dem Embedded Linux Entwickler sehr viele Möglichkeiten bietet - insbesondere mit Hardwareerweiterungen ließen sich mit dem NGW ziemlich interessante Projekte realisieren.

Jedoch wäre besonders für die Hardwareerweiterungen eine Möglichkeit zum Anfertigen von Platinen innerhalb der FH sehr wünschenswert - denn die Anzahl der SMD-Bauteile nimmt stetig zu und nach meinem persönlichem Empfinden sind sehr viele Projekte (z.B. Audio-Codec) daran gescheitert.

Auch wenn die Hardwareerweiterungen in meinem Fall auch noch nicht funktionieren, habe ich mein Grundziel erreicht - ich kann den NGW100 ständig am Netz lassen und ihn zumindest als *Instant Messenging Anrufbeantworter* verwenden.

Im Laufe der Semesterferien werde ich jedoch garantiert noch an diesem Board weiterarbeiten.

## 4. Fazit

Die diversen Versuche und Projekte im Umfeld von Embedded Linux waren sehr interessant und lehrreich - während dieser Studienarbeit habe ich sehr viel über Kernel-Internas, das sysfs und CrossCompiling mit all seinen Tücken gelernt.

Auch wenn die Arbeit manchmal sehr frustrierend war, hat die Arbeit mit den beiden Boards sehr viel spaß gemacht, was man wahrscheinlich an meinem Engagement im Bezug auf den Umfang der Projekte auch gemerkt hat.

Insbesondere die Erfolgserlebnisse bei der hardwarenahen Softwareentwicklung machen diese kleinen Tiefs mehr als wieder wett - denn wenn es dann nach viel harter Arbeit klappt freut man sich riesig über seinen Erfolg - selbst wenn es sich dabei nur um ein kleines e auf einem LCD-Display handelt!